Scan and Join Optimization by Exploiting Internal Parallelism of Flash-Based Solid State Drives

Wenyu Lai, Yulei Fan, and Xiaofeng Meng

Renmin University of China, Beijing, China
{xiaolai913,fyl815,xfmeng}@ruc.edu.cn

Abstract. Nowadays, flash-based solid state drives (SSDs) are gradually replacing hard disk drives (HDDs) as the primary non-volatile storage in both desktop and enterprise applications because of their potential to speed up performance and reduce power consumption. However, database query processing engines are designed based on the fundamental characteristics of HDDs, so they may not benefit immediately from SSDs. Previous researches on optimizing database query processing on SSDs have mainly focused on leveraging the high random data access performance of SSDs and avoiding slow random writes whenever possible. However, they fail to exploit the rich internal parallelism of SSDs. In this paper, we focus on exploiting rich internal parallelism of SSDs to optimize scan and join operators. Firstly, we detect internal parallelism of SSDs seemed as black boxes. Then we propose a parallel table scan operator called ParaScan to take full advantage of internal parallelism of SSDs. Based on ParaScan, we also present an efficient parallel join operator called ParaHashJoin to accelerate database query processing. Experimental results on TPC-H datasets show that our ParaScan on SSD significantly outperforms the traditional table scan on SSD by 1X, and ParaHashJoin is 1.5X faster than traditional hash join operator especially when join selectivity is small.

Keywords: Flash-based SSDs, Internal Parallelism, Query Processing.

1 Introduction

Flash-based solid state drives (SSDs), as a new type of non-volatile storage, are gradually replacing the central role of traditional magnetic hard disk drives (HDDs). More and more portable devices are equipped with SSDs to get excellent performance such as MacBook Air, Play Station and so on. Compared to HDDs, SSDs provide faster access speed, lower power consumption, lighter weight, smaller size and better shock resistance. In addition to these advantages, SSDs also have rich internal parallelism which provides us a chance to improve I/O bandwidth by doing parallel processing on them [1, 2]. Due to their potential to speed up applications and reduce power consumption, SSDs are expected to gradually substitute HDDs as mass storage media in large data centers.

The development of flash-based SSDs is also attracting researchers' interests to redesign various aspects of DBMS internals for SSDs such as storage management,

J. Wang et al. (Eds.): WAIM 2013, LNCS 7923, pp. 381–392, 2013.
© Springer-Verlag Berlin Heidelberg 2013

query processing and so on. Traditional query processing algorithms are mainly designed according to the mechanical traits of the disk, so they may benefit less or even nothing when SSDs are used as a simple drop-in replacement for disk for data analysis workloads in database [3]. Thus it is necessary to redesign the query processing algorithms to take full advantages of SSDs.

Scan and join are two important operators in DBMS. Scan operators are basic physical operations in database system and they can mainly divide into two categories: table scan and index scan [4]. Table scan reads data block one by one sequentially, which is good suit for HDDs. Lots of query processing operators often need to cooperate with table scan to accomplish a query such as sort, aggregation and join. Join operators are also basic operations in database system, but they are more complex. It is well-known that join operations can be expensive and can play a critical role in determining the overall performance of the DBMS. There are three classic ad hoc join algorithms in traditional DBMS, namely: nested loops join, sort-merge join and hash join [4]. In this paper, we mainly explore the optimization of table scan and hash join.

Due to rich internal parallelism of SSDs, it is possible for us to process multiple I/O requests at the same time on SSDs, that may offer us excellent IOPS (Input/Output Operations Per Second). However, the outstanding random I/O performance of SSDs will remain only a potential performance specification, unless DBMSs take advantage of internal parallelism and fully utilize the high IOPS.

In this paper, we investigate query processing methods that are better suited for the characteristics of SSDs, especially SSDs' internal parallelism. In particular, we focus on speeding up scan and join operations over tables stored on SSDs. Towards this goal, we make the following contributions.

- We detect and examine internal architecture of different kinds of SSDs and then propose a novel parallel table scan operator called ParaScan to take full advantage of internal parallelism of SSDs.
- Based on ParaScan, we present ParaHashJoin, an efficient parallel hash join operator which makes full use of internal parallelism of SSDs to accelerate join processing in a query plan.
- Experiment evaluation results on TPC-H datasets demonstrate that the proposed ParaScan and ParaHashJoin operators reduce the execution time of scan and join effectively.

The rest of the paper is organized as follows. In Section 2, we give the related work and compare our study with them. Section 3 introduces internal parallel architecture of SSDs at first and then discusses how to utilize this internal parallelism and why we should detect SSD internals. After that, we propose a parallel table scan operator and a parallel join operator in Section 4 and Section 5 respectively. Section 6 describes experimental results on TPC-H datasets, and we conclude in Section 7.

2 Related Work

In order to achieve high bandwidth and better IOPS, most modern SSDs adopt multi-channel and multi-way architecture and flash memory controller can access flash chips in parallel [5]. Therefore, we need to understand the impact of this parallelism inside SSDs to improve data processing performance. Chen et al. [2] studied on how to uncover internal parallelism features of SSDs and revealed that exploiting internal parallelism can significantly improve I/O performance. In the study of [6], researchers focused on finding an efficient way to generate parallel I/Os to access SSDs. By assessing different methods to create parallel I/O, authors suggest a new I/O request method and design a new B+-tree variant called PIO B-tree to exploit internal parallelism of SSDs. Other studies [7, 8] tried to improve SSD internal architecture design in order to provide more I/O parallelism inside SSDs.

There are several works that investigate in database query processing techniques on flash-based SSD [9, 10, 11, 12]. Graefe et al. [10], [11] focus on data structures and algorithms that leverage fast random reads to speed up selection, projection and join operations in query processing. They explore the impact of new page layouts on SSDs and propose FlashScan, FlashJoin and RARE-join algorithms. Another typical work is DigestJoin [12] which focuses on exploring the possibility of further improving non-index join algorithms by reducing intermediate results and utilizing fast random reads of SSDs. Different from previous studies, we try to optimize scan and join performance by exploiting internal parallelism of flash-based SSDs.

3 Internal Parallelism of SSD

In this section, we introduce internal parallel architecture of SSDs at first and then discuss how to utilize this rich internal parallelism and why we should detect SSD internals.

3.1 Internal Parallel Architecture

A flash-based SSD internal architecture is presented in Fig. 1. The *host interface* is used to connect with the host and its common interface type is SATA. *SSD controller* is the brain of an SSD which is in charge of executing I/O requests and issues commands to flash memory packages via the *flash controller*. Inside SSD, there is a *RAM buffer* to hold the mapping table and other metadata. A flash-based SSD implements internal parallelism by adopting multiple channels which be shared by a set of flash memory packages. Each channel can be operated independently and simultaneously while operations on flash memory packages attached to the same channel can also be interleaved, so the bus utilization can be optimized [5, 13]. By examining the internal architecture of SSDs, we can find that there are two typical levels of parallelism which are channel-level parallelism and package-level parallelism. Such rich internal parallelism provides us an opportunity to improve the performance of applications on SSDs.

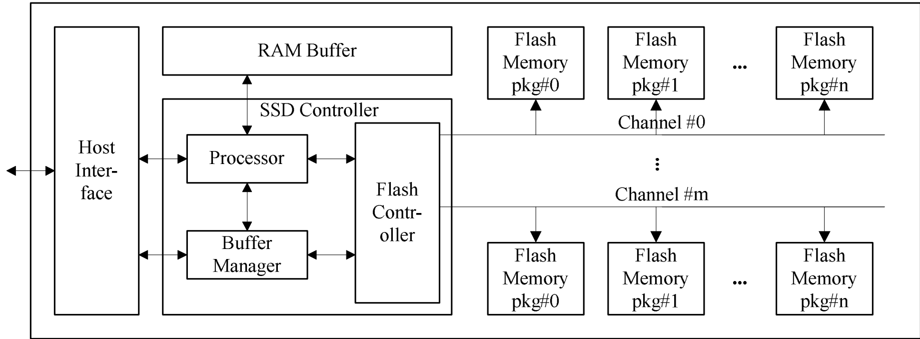

Fig. 1. Flash-based SSD internal architecture

3.2 How to Utilize Internal Parallelism

In order to utilize channel-level parallelism and package-level parallelism, multiple I/O requests designated to different flash memory packages spanning several channels should be submitted to SSDs at the same time whenever possible. In this way, native command queuing (NCQ) mechanisms of the host interface can generate favorable I/O patterns to the internal architecture [14]. In this paper, we mainly use multi-thread processing technique to produce multiple parallel I/Os at the same moment.

3.3 Detecting SSD Internals

To make better use of internal parallelism of SSDs, it is necessary to know some key architectural features of an SSD. For example, knowing the number of channels in an SSD, we can set a proper concurrency level and avoid over-parallelization. However, it is hard to obtain such key architectural information because these details are often regarded as critical intellectual property of SSD manufacturers. Therefore, we have to do some detecting work to know about the SSD internals.

We select two representative and state-of-the-art SSDs for research. One is built on multi-level cell (MLC) flash memories, which is designed for the mass storage market, while the other is a high-end product built on faster and more durable single-level cell (SLC) flash memories. For simplicity, we refer to these two SSDs as SSD-S and SSD-M respectively. More details about these two SSDs are shown in Table 1.

Table 1. Specification of Observed SSDs

	SSD-S	SSD-M
Manufacturer	Intel	Intel
Flash Memory	SLC	MLC
Capacity (GB)	32	160
Page Size (KB)	4	4
Interface Type	SATA	SATA
NCQ	32	32

Despite various implementations, most designers of SSDs try to optimize performance essentially in a similar way that is evenly distributing data accesses to maximize resource usage. Base on some open documents, Chen et al. [2] define an abstract model to characterize the internal organization of SSD. In that model, a *domain* is a set of flash memories that share a specific set of resources (e.g. channels). A domain can be further partitioned into *sub-domains* (e.g. packages). A *chunk* is a unit of data that is continuously allocated within one domain. Chunks are interleavingly placed over a number of domains. Guided by the model and the detecting method introduced in [2], we detected the chunk size and the number of domains on SSD-S and SSD-M respectively. The detecting results are shown in Table 2.

Table 2. Detecting Results

	SSD-S	SSD-M
Chunk Size (KB)	4	16
Domains Number	20	20

4 ParaScan

In this section, we first give an overview of ParaScan, a parallel table scan operator, which is designed guided by internal parallelism of flash-based SSDs and then describe its main components including domain scan and multi-domain parallel scan in the following subsections.

4.1 ParaScan Overview

Despite various implementations, most SSDs adopt a RAID-0 like striping data storage mechanism as is shown in Fig. 2. The striped chunks of the domains are mostly placed in consecutive LBA (Logical Block Address) regions. In the striped domains, the write interleaving technique enhances the write performance by avoiding the shared data-bus channel competition and by interleaving data transfers while other domains are writing the already transferred data. This data storage policy provides us a chance to do parallel table scan and maximize resource usage by distributing data accesses to different domains. Therefore based on that good nature, we propose a parallel table scan called ParaScan to improve the efficiency of table scan.

As shown in Fig. 2, the basic operation of ParaScan is *domain scan*. Domain scan read data chunks one by one from a single domain and then put them into the buffer. To reduce the potential performance loss caused by sharing resources, each domain scan maintains a small scan buffer called ScanBuffer. In this way, multiple domain scan can be executed in parallel without any interference. Multiple domain scan and multiple ScanBuffers compose the *multi-domain parallel scan*. As domain is a parallel unit of solid state drive, when we want to write data to SSDs, we should consider that all data pages of a relational table should be distributed into different domains as much as possible to make full use of internal parallelism of SSDs.

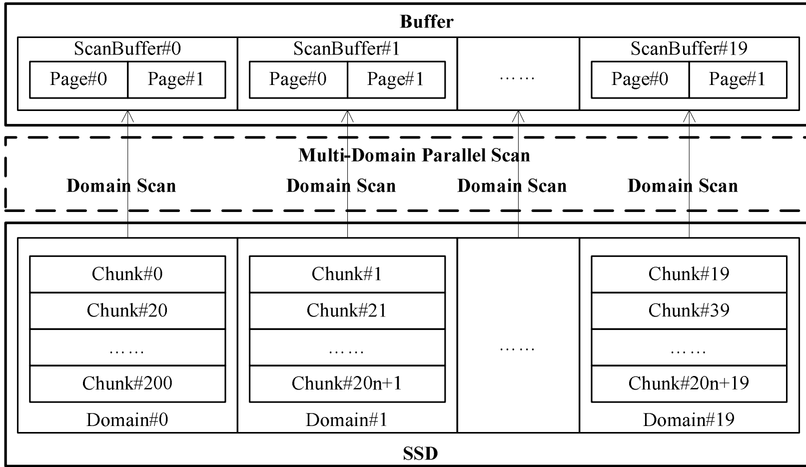

Fig. 2. Overview of ParaScan

4.2 Domain Scan and Multi-domain Parallel Scan

Domain scan is similar to traditional database table scan. Traditional database table scan read data blocks of a relational table one by one, and then put them into a buffer for processing. The different between domain scan and traditional table scan is that domain scan read data chunks one by one which all come from the same domain, and then put them into ScanBuffer. Due to the size of data stored in the same domain is usually exceed ScanBuffer size, the processed contents need to be replaced in order to read unprocessed data pages. Because here we do not consider the data write operation, the management of the ScanBuffer becomes relatively easier, we only need to cover the processed contents directly to read unprocessed data.

We implement multi-domain parallel scan by multi-thread processing. ParaScan operator generates multiple threads to do scan operation and each of them is in charge of one or multiple domain scan. Entire scan buffer is also divided into several ScanBuffers so that each scan thread can use one ScanBuffer. The performance of multi-domain parallel scan depends on the concurrency level which is not only related to the number of domains but also the maximal number of physical threads supported by the processor and the maximal queue depth supported by the SSD.

5 ParaHashJoin

Based on ParaScan operator, we present ParaHashJoin in this section. We first give an overview of our join operator in Section 5.1 and then describe its main components including ParaHash, MiniJoin and Fetch in the following subsections.

5.1 ParaHashJoin Overview

ParaHashJoin is a parallel hash join operator tuned for solid state drives. First, it parallels the join operation as much as possible to make use of internal parallelism of SSDs. Moreover, ParaHashJoin also learns some important ideas from previous researches to take advantage of the fast random reads of SSDs. It uses late materialization strategy to avoid processing unneeded attributes and postpone retrieving projected attributes until absolutely necessary. In this paper, we mainly consider a two-way equi-join implement of ParaHashJoin, and the multi-way join implement will continue to be researched in the future.

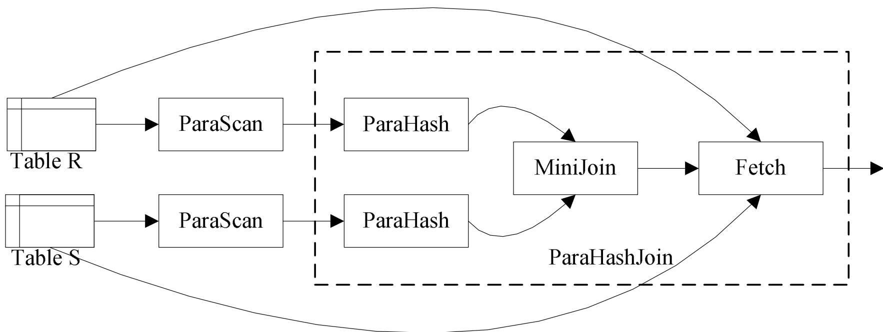

Fig. 3. Overview of ParaHashJoin

Each ParaHashJoin in two-way equi-join consists of three phases, ParaHash phase, MiniJoin phase and Fetch phase, as shown in Fig. 3. The ParaHash phase builds a hash table in parallel on the join attributes and the row-ids (RIDs) of the participating rows from input relational table. A RID specifies the page and offset within the page for that row. ParaHashJoin uses a late materialization strategy in which the Fetch phase retrieves the needed attributes using the RIDs specified in join results produced by MiniJoin. This approach offers some important benefits over traditional joins which use an early materialization strategy.

First, based on ParaScan, ParaHashJoin can hash table records belong to different domains in parallel which not only make full use of internal parallelism of SSDs but also the ability of multi-core processor. Second, ParaHashJoin is more memory efficient. By using late materialization strategy, ParaHashJoin greatly reduces the amount of data needed to be read to compute the join result. Moreover, when multiple passes are needed, it incurs lower partitioning cost than traditional joins.

However, to get these benefits, we have to pay more CPU cost to do multi-thread processing and more cost of random reads for retrieving the other projected attributes in the fetch phase. But we show that this tradeoff is worthwhile to do a join on SSDs in the experimental section.

5.2 ParaHash

In ParaHash phase, we use the same technique as ParaScan to implement parallel processing. Fig. 4 is a sketch of ParaHash in which buffer is divided into two regions, scan area and hash area. After ParaScan a table, we generate multiple hash threads to calculate the hash values of records that have been scanned into ScanBuffers, and then put their join attributes and RIDs into corresponding hash buckets in parallel. Each hash thread is in charge of one ScanBuffer. To mitigate the cost of calculations, here we use a simple hash function which applies a fast bit operator, as shown in Eq. (1). In this equation, *join_attr* represent the join attribute value of a record to be hashed and *B* is the number of hash buckets which should be the power of 2.

$$hash_value = join_attr \ \& \ (B - 1) \tag{1}$$

As we implement ParaHash by multi-thread processing, some threads may hash different records into the same bucket at the same time. Thus, each bucket should maintain a lightweight lock to do concurrency control. Moreover, we build a bitmap for each bucket to quickly judge whether hash index records with specified join attribute exist in the bucket.

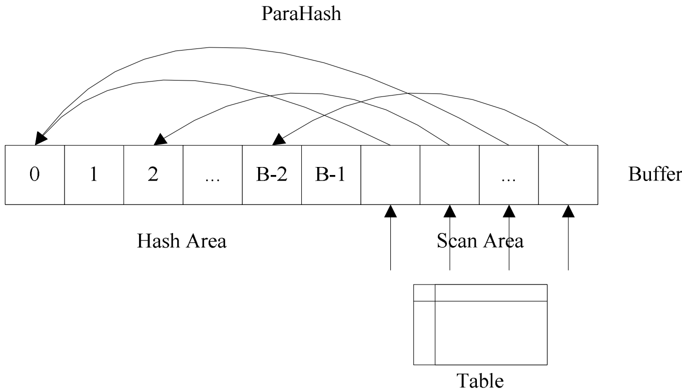

Fig. 4. Sketch of ParaHash

5.3 MiniJoin

After ParaHash table R and ParaHash table S, we have already get two hash tables. Then, in the MiniJoin phase, we read each hash bucket into memory and generate the join results, in the form of $\{join_attr, RID_R, RID_S\}$. The MiniJoin results may be written to the SSD sequentially if it is larger than the memory size.

In the case memtioned above, we need two pass to generate the MiniJoin results, one pass for ParaScan and one pass for MiniJoin. But if there is enough memory to hold the hash table of table R, the smaller table, we only need one simple pass. It first ParaScan and ParaHash table R, then it ParaScan table S. After that, in the MiniJoin phase, it can directly probe the hash table and produce MiniJoin results.

5.4 Fetch

However, the outputs of the MiniJoin phase are the incomplete join results which only tell us which records satisfy the join. Therefore, in the Fetch phase, we should fetch the necessary attributes using the RIDs specified in the join index to generate the final join results.

In this phase, an efficient fetching strategy is very important for ParaHashJoin as it minimizes the number of page accesses when fetching the needed attributes from the original tables. A straightforward strategy is to fetch the related pages specified by RIDs as soon as they are produced in the MiniJoin phase. For each join result of MiniJoin, ParaHashJoin fetch the needed data pages in the buffer pool or retrieves them from peripheral storage, and then generate the final join result. This approach is reasonable when all data pages needed to generate the result can fit in memory. However, when available memory is insufficient, this approach may result in reading some pages multiple times because the RIDs in the join index are usually unordered. Thus, the larger the join result, the higher is the cost of reloading pages. TID hash joins [15] use this approach which is their biggest weakness.

We adopt a sort-based fetching strategy inspired by DigestJoin [11] to avoid reloading pages as much as possible. Before fetching the matching pages according to RIDs in MiniJoin results, we sort MiniJoin results based on the RIDs of outer table at first. Then we begin to load needed pages to produce final join results according to the sorted MiniJoin results. In this strategy, we need to pay more cost to execute this sort, but we show that this payment is worthwhile in the experimental section.

6 Experiment Evaluation

In this section, we present experimental evaluations of ParaScan and ParaHashJoin. We first describe the experimental setup in Section 6.1. Then we present experimental results of ParaScan and ParaHashJoin on TPC-H datasets in Section 6.2 and Section 6.3 respectively.

6.1 Experimental Setup

Our experiments all run on a HP PC with Ubuntu 12.10 operating system. This platform is equipped with Intel Core i5-2400 @ 3.10GHz processor which is of four cores and supports four physical threads. In addition, it is equipped with 8G DDR3 memory, a 500G 7400rpm SATA3 Seagate magnetic disk and two kinds of SSDs as shown in Table 1. In order to make use of SSDs' rich internal parallelism, we need to enable AHCI (Advanced Host Controller Interface) mode by setting the BIOS.

For our experiments, we implement two operators, ParaScan and ParaHashJoin, by multi-thread C programming. To avoid the interference from the buffer of file system, we read/write files in DirectIO mode and align the memory manually. The test dataset is taken from the TPC-H benchmark. In particular, we use CUSTOMER table and ORDERS table to do scan and join. CUSTOMER table has 1.5 million rows in total size of about 256MB, while ORDERS table has 150 million rows in total size of about 2GB. In following subsections, we have executed scan operations on ORDERS table and join operations between CUSTOMER table and ORDERS table.

6.2 ParaScan Evaluation

We executed a table scan on ORDERS table by using ParaScan in our HDD, and two kinds of SSDs respectively. Fig. 5 compares the performance of scans as we vary the number of parallel scan threads. For simplicity, we define the symbol ParaScan-N to represent running ParaScan with N parallel threads. As traditional table scan reads data block one by one sequentially in a single-thread mode, it is equal to ParaScan-1. As our CPU is of 4 physical threads and we have known that SSD-S and SSD-M both have 20 domains, we tested the performance of ParaScan under 1, 4, 10, 20 and 30 parallel threads.

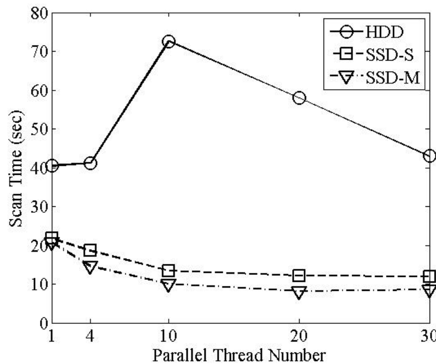

Fig. 5. Performance comparation of traditional table scan and ParaScan

According to the results shown in Fig. 5, we learn that ParaScan doesn't suit for HDD due to its magnetic characteristics. However, the scan time of ParaScan on two kinds of SSDs both decrease apparently with the increasing number of parallel threads. When the number increases to 30, the performance of ParaScan on SSDs still has an improvement but this improvement is very little. That is because the cost to switch among parallel threads also increases. Fig. 5 shows that, in best case, the scan time of ParaScan on SSD is only half of time of traditional table scan on SSD and only a quarter of time of the traditional table scan on HDD.

6.3 ParaHashJoin Evaluation

In our experiments, we compare the performance of ParaHashJoin with two kinds of join algorithms. One is TradHashJoin which refers to traditional hash join using table scan and the other one is NewHashJoin which represents traditional hash join using ParaScan. We mainly consider the equi-join of two relations CUSTOMER and ORDERS on a single join attribute and use *selectivity* to refer to the percentage of results size. The number of parallel threads in ParaScan and ParaHashJoin is set to be 20. Due to the length limit of this paper, we only present results of experiments on SSD-S but we also conducted experiments on SSD-M and got consistent results.

In the first experiment, we vary the selectivity of the join result from 0.01% (1500 rows) to 10% (1.5 million rows). The amount of memory allocated to each join is 8MB so that all of the joins are executed in two pass. As shown in Fig. 6, the execution times of all algorithms increase with the growth of the selectivity. That is because higher selectivity can lead to larger intermediate results and hence higher partitioning costs. However, ParaHashJoin suffers the least when join selectivity is small since it only partitions the join attributes and take full advantage of internal parallelism of SSD at the same time.

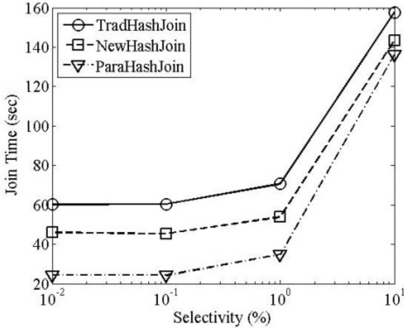

Fig. 6. Performance comparison of join operators under different selectivity

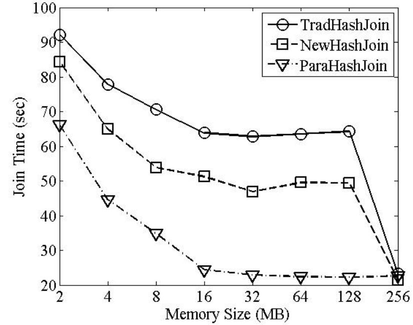

Fig. 7. Memory impact on joins

To evaluate memory impact on joins, we set join selectivity at 1% and then vary the amount of memory allocated per join form 2MB to 256MB. Fig. 7 presents our results. We can see that both TradHashJoin and NewHashJoin require 256MB to execute the join between CUSTOMER and ORDERS in one pass while ParaHashJoin only requires 16MB. Fig. 7 shows that, in best case, ParaHashJoin is 1X faster than NewHashJoin and 1.5X faster than TradHashJoin. From these results, we can see that it is worthwhile to pay extra CPU cost for multi-thread processing and extra cost of random reads for retrieving projected attributes in the fetch phase.

7 Conclusion

In this paper, we detect and examine internal architecture of different kinds of SSDs and then based on rich internal parallelism of SSDs, we propose ParaScan and ParaHashJoin operators to accelerate scan and join processing on SSDs. ParaScan takes full advantage of internal parallelism of SSDs through multi-thread processing techniques. Based on ParaScan, ParaHashJoin not only parallels the join operation as much as possible but also applies some important ideas from previous researches to take advantage of the fast random reads of SSDs. The experimental results show that ParaScan operator is 1X faster than traditional table scan on SSD and 2X faster than traditional table scan on HDD in best case. And the execution of traditional hash join algorithm on SSD is 3 times longer than ParaHashJoin. These fully demonstrate the superiority of ParaScan and ParaHashJoin.

Acknowledgements. This research was partially supported by the grants from the Natural Science Foundation of China (No. 60833005, 61070055, 91024032, 91124001); the National 863 High-tech Program (No. 2012AA010701, 2013AA013204); the Fundamental Research Funds for the Central Universities, and the Research Funds of Renmin University(No. 11XNL010).

References

1. Chen, F., Koufaty, D.A., Zhang, X.: Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In: SIGMETRICS, pp. 181–192 (2009)
2. Chen, F., Lee, R., Zhang, X.: Essential Roles of Exploiting Parallelism of Flash Memory based Solid State Drives in High-Speed Data Processing. In: HPCA, pp. 266–277 (2011)
3. Lee, S.-W., Moon, B.: Design of flash-based DBMS: An in-page logging approach. In: SIGMOD, pp. 55–66 (2007)
4. Ramakrishnan, R., Gehrke, J.: Database Management Systems, 3rd edn. McGraw Hill (2002)
5. Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J.D., Manasse, M., Panigrahy, R.: Design Tradeoffs for SSD Performance. In: USENIX, pp. 57–70 (2008)
6. Roh, H., Park, S., Kim, S., Shin, M., Lee, S.-W.: B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives. Proceedings of the Very Large Data Base (VLDB) Endowment 5(4), 286–297 (2012)
7. Hu, Y., Jiang, H., Feng, D., Tian, L., Luo, H., Zhang, S.P.: Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity. In: ICS, pp. 96–107 (2011)
8. Park, S.Y., Seo, E., Shin, J.Y., Maeng, S., Lee, J.: Exploiting internal parallelism of flash-based SSDs. Computer Architecture Letters 9(1), 9–12 (2010)
9. Do, J., Patel, J.M.: Join processing for flash SSDs: remembering past lessons. In: DaMoN, pp. 1–8 (2009)
10. Shah, M.A., Harizopoulos, S., Wiener, J.L., Graefe, G.: Fast scans and joins using flash drives. In: DaMoN, pp. 17–24 (2008)
11. Tsirogiannis, D., Harizopoulos, S., Shah, M.A., Wiener, J.L., Graefe, G.: Query processing techniques for solid state drives. In: SIGMOD, pp. 59–72 (2009)
12. Li, Y., On, S.T., Xu, J., Choi, B., Hu, H.: DigestJoin: Exploiting Fast Random Reads for Flash-based Joins. In: MDM, pp. 152–161 (2009)
13. Dirik, C., Jacob, B.: The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device, architecture, and system organization. In: ISCA, pp. 279–289 (2009)
14. Lee, S.-W., Moon, B., Park, C.: Advances in flash memory SSD technology for enterprise database applications. In: SIGMOD, pp. 863–870 (2009)
15. Marek, R., Rahm, E.: TID hash joins. In: CIKM, pp. 42–49 (1994)